Distributed Framework for Dynamic Telescope and Instrument Control

Troy J. Ames[a], Lynne Case[b]
[a]NASA Goddard Space Flight Center, [b]Aquilent, Inc.

ABSTRACT

Traditionally, instrument command and control systems have been developed specifically for a single instrument. Such solutions are frequently expensive and are inflexible to support the next instrument development effort. NASA Goddard Space Flight Center is developing an extensible framework, known as Instrument Remote Control (IRC) that applies to any kind of instrument that can be controlled by a computer. IRC combines the platform independent processing capabilities of Java with the power of the Extensible Markup Language (XML). A key aspect of the architecture is software that is driven by an instrument description, written using the Instrument Markup Language (IML). IML is an XML dialect used to describe graphical user interfaces to control and monitor the instrument, command sets and command formats, data streams, communication mechanisms, and data processing algorithms.

The IRC framework provides the ability to communicate to components anywhere on a network using the JXTA protocol for dynamic discovery of distributed components. JXTA (see http://www.jxta.org) is a generalized protocol that allows any devices connected by a network to communicate in a peer-to-peer manner. IRC uses JXTA to advertise a device's IML and discover devices of interest on the network. Devices can join or leave the network and thus join or leave the instrument control environment of IRC.

Currently, several astronomical instruments are working with the IRC development team to develop custom components for IRC to control their instruments. These instruments include: High resolution Airborne Wideband Camera (HAWC), a first light instrument for the Stratospheric Observatory for Infrared Astronomy (SOFIA); Submillimeter And Far Infrared Experiment (SAFIRE), a Principal Investigator instrument for SOFIA; and Fabry-Perot Interferometer Bolometer Research Experiment (FIBRE), a prototype of the SAFIRE instrument, used at the Caltech Submillimeter Observatory (CSO). Most recently, we have been working with the Submillimetre High Angular Resolution Camera IInd Generation (SHARCII) at the CSO to investigate using IRC capabilities with the SHARC instrument.

Keywords: astronomy, infrared astronomy, XML, instrument: description, telescope: control, SOFIA, NASA: GSFC, Java, IML

1. INTRODUCTION

NASA Goddard Space Flight Center's Instrument Remote Control (IRC) project is an ongoing effort led by the Advanced Architectures and Automation Branch (Code 588). The IRC project supports NASA's mission by defining an adaptive framework that provides robust, interactive, distributed control and monitoring of remote instruments. The IRC framework will eventually enable trusted astronomers around the world to easily access infrared instruments (e.g., telescopes, cameras, and spectrometers) located in remote environments such as the South Pole, high mountaintops, or an airborne observatory aboard a Boeing 747. The IRC framework will enable astronomers, instrument designers, hardware engineers, and other scientists to define new instruments, control these instruments remotely, and monitor vital instrument telemetry over an intranet or for trusted users, the Internet.

The IRC framework will be applied as an operational solution for the High-resolution Airborne Wideband Camera (HAWC) and the Submillimeter And Far Infrared Experiment (SAFIRE, a spectrometer) on the Stratospheric Observatory for Infrared Astronomy (SOFIA). The IRC architecture was also used to develop the control and data acquisition software for one of the proposed detector concepts for the Spectral and Photometric Imaging REceiver (SPIRE), a focal plane instrument for the European Space Agency's (ESA) Far Infrared Space Telescope (FIRST).

IRC is a platform independent framework, designed to be generic and extensible so that it can be applied to any instrument capable of being computer controlled. In order to design an extensible and flexible architecture, the established goals of the IRC project are to:

- Provide as much platform independence as possible.
- Create a system that is easy to develop, maintain, and extend.
- Explicitly promote reuse by design and utilize emerging technologies that facilitate software reuse.
- Greatly reduce the implementation time for facility instruments, which must be reliable, robust, state-of-the-art, and easily used by scientists other than the instrument's designers.
- Clearly define the interface between hardware and software engineers.
- Facilitate multiple iterations of the instrument description during design and implementation by means of a software architecture that is readily adaptable to such changes.
- Cleanly separate implementation from description.

2. INSTRUMENT MARKUP LANGUAGE

Working with instrument engineers and scientists in the astronomy domain (and infrared instruments in particular), The Instrument Markup Language (IML) was developed as a means to describe an instrument. IML is a vocabulary of the Extensible Markup Language (XML), a W3C standard. An XML schema enables an XML parser to validate the XML files, thereby guaranteeing that the instrument description is complete and correct according to the content constraints of the IML schema definition. A key aspect of the IRC architecture, implemented in Java, is that the software is driven by the IML instrument description. The attributes of an instrument that can be described by IML include:

- Instrument subsystems
- Logical command set
- Command arguments (including data types, valid values/ranges, and units)
- Command formats
- Logical data streams (e.g., science data, housekeeping, command responses)
- Data field types (including data types, valid values/ranges, and units)
- Data formats
- Communication mechanisms
- Documentation

Although at this stage in the evolution of IML it is primarily the software engineers who are writing the descriptions, the IRC team envisions the hardware engineers taking on this task. Not only do hardware engineers know the instrument details the best, but they traditionally provide significant contributions to formal Interface Control Documents (ICD). The IML documents can serve a similar role – that is, communicating the intricate details of an instrument's operation – in a much more structured, formal, and easily manipulated way. Using IML in ICDs will require that hardware engineers are provided with software tools that hide the details of XML syntax. By utilizing an XML editor with a sufficiently self-explanatory IML Schema, hardware engineers can be guided through the process of instrument specification by noting what elements are applicable in a certain context based on the schema's content model. The IRC Configuration Editor application is an initial implementation of this IML specific editor.

Although IML is currently applied primarily to astronomical instruments, the key aspects of our approach to instrument description and control apply to many domains, from medical instruments (e.g., microscopes) to printing presses to machine assembly lines. Due to the extensible nature of XML, we can easily imagine dialects of IML for various domains, such as the Astronomical Instrument Markup Language (AIML). Currently, there is no separate AIML Schema. The generic IML Schema provides all of the necessary functionality. But a separate AIML Schema could be created if needs were identified which were not met by the general instrument description solution. The IML (or AIML) Schema can be extended to support new instrument requirements, often without requiring changes to previously written instrument descriptions.

3. INSTRUMENT REMOTE CONTROL ARCHITECTURE

The IRC framework is implemented in Java. Figure 1 illustrates the high level architecture of the IRC framework. The IML file drives the behavior of many general Java objects.

A default Graphical User Interface (GUI) is provided with IRC. It automatically creates a command panel upon reading the IML instrument description. This default GUI provides the means to issue all of an instrument's (and its subsystems') commands. Since the IML file describes all of the arguments (including the arguments' data types and valid values), the GUI can present a command window that enables a user to issue valid commands.

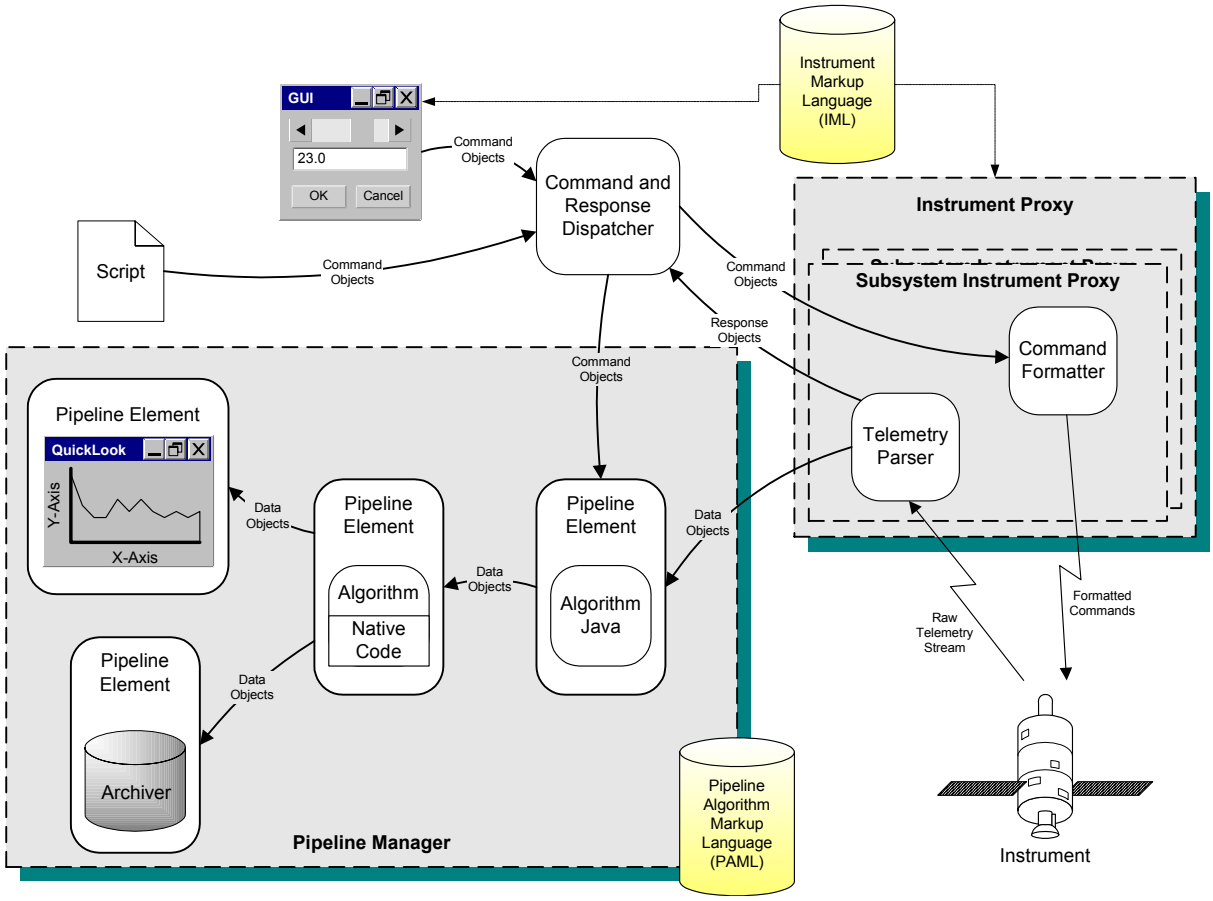

Figure 1: High Level IRC Framework Architecture

The Instrument Proxy creates objects that know how to communicate directly with the instrument. The IML instrument description specifies the communication mechanism (e.g., TCP/IP, Serial) and the formatting rules for commands. IML can describe a hierarchy of sub-instruments, and each subsystem (sub-instrument) may use a different communication mechanism or protocol. For example, one subsystem may have a TCP/IP interface with binary commands and another subsystem may have an RS232 interface with ASCII commands. Each subsystem in the IML instrument description is represented by its own Instrument Proxy, which receives command objects, formats them according to the rules specified in the IML file, and then sends them to the actual instrument.

The flow of command and data objects through the system is managed using a Publish/Subscribe pattern. Subscribers register with publishers for objects that they are interested in - either for all objects published, or for just those objects that match particular criteria. This facilitates a dynamic and distributed flow of control. Instrument proxies subscribe to

receive commands from a singleton Command Dispatcher. GUIs can come and go based on user demands. A GUI publishes commands via the Command Dispatcher, which in turn publishes them to the interested instrument proxies. This isolates the proxies from the transient nature of the GUIs.

The Instrument Proxy also creates a Parser object for each instrument port that will be receiving telemetry as defined in the IML file. The parsing rules described in the IML file define how raw data is parsed into Data Objects. Many of the generic framework objects allow the IML instrument description to specify instrument-specific delegates. For example, the SPIRE instrument had four subsystems, which produced six different telemetry streams. The formats of five of the telemetry streams could be described such that the generic telemetry parsing engines could parse those streams directly. However, the generic parsing engine could not handle the science data stream. The format of the data was extremely complex, and the performance requirements were significant – the software had to handle data rates of up to 30 MBps. A parser highly optimized for this data was implemented and tuned to the demands of the SPIRE science data stream. The parsing delegate was specified in the IML instrument description. At runtime, the Instrument Proxy created an instance of this delegate, and plugged it into the science data port. Java's ability to load classes dynamically makes it straightforward for the generic framework to perform instrument-specific operations without prior having knowledge of the delegate classes.

Figure 1 shows the data analysis pipeline in the lower right. The data analysis pipeline allows the users to specify algorithms that operate on the data in real time for quick look analysis of the data during operations. IRC is delivered with a set of algorithms, but users are allowed to specify new algorithms at run-time. Again, this is accomplished through Java's ability to dynamically load classes at run-time. Section 5 describes the data analysis pipeline in more detail.

3.1. IML Examples

This section includes some examples that illustrate how a typical instrument is described using IML. Note that much of the detail is omitted from these examples for presentation purposes. Consider an instrument named HAWC that contains four subsystems: Detector, ADR, Optics, and Telescope.

```
<Instrument id="HAWC">
     <Instrument id="Detector"> … </Instrument>
     <Instrument id="ADR"> … </Instrument>
     <Instrument id="Optics"> … </Instrument>
     <Instrument id="Telescope">
          <!-- Command and Data descriptions go here (see Figure 5) -->
          <Port name="Telescope" portType="TCP" encodingType="RegExp">
               <Parameter name="hostname" value="telescope.gsfc.nasa.gov"/>
               <Parameter name="number" value="4055"/>
               <Parameter name="serverPort" value="true"/> >
               <!-- Command and Data formats go here (see Figure 7) -->
          </Port>
     </Instrument>
</Instrument>
```

Figure 2: IML Description of Instrument and Subsystems

A subsystem can have multiple ports; for example, there may be a need for a commanding port and a telemetry port. The IML fragment in Figure 2 states that the Telescope subsystem has a single TCP port for commanding, and by examining the port element we see that commands are formatted using a Regular Expression Formater (RegExp). In addition to the TCP port, the framework supports several other communication mechanisms such as RS232 and DMA. The architecture allows support for additional protocols to be added easily.

Now let's examine a sample command from the Telescope subsystem, see Figure 3. The Secondary command has five arguments. The Chopper Throw argument is specified as a float and is the only required argument for this command. It is represented by a text field in the GUI, see Figure 4. Since the argument has a range constraint specified the GUI checks that the user input is within the range and of the correct type, displaying an error dialog if it is not. The Chop Qualifier argument is of type String and specifies a List Constraint for the acceptable valid values. The GUI represents the valid choices from the List Constraint as a pop-up menu.

```
<Command name="Secondary" abbreviation="secondary"
          docHelp="This command controls the chopping
          secondary mirror.">
          <Field name="Chopper Throw" type="Float"
               required="true"
               default="120.0" instrumentUnit="arcsec"
               docHelp="Specifies the separation of the
               beams on the sky">
               <RangeConstraint low="0.0" high="540.0"/>
          </Field>
          <Field name="Chop Frequency" type="Float"
               required="false"
               default="4.0" instrumentUnit="Hz">
               <RangeConstraint low="0.0" high="5.0"/>
          </Field>
          <Field name="On Tolerance" type="Float"
               required="false"
               default="10.0" instrumentUnit="arcsec"/>
          <Field name="Off Tolerance" type="Float"
               required="false"
               default="10.0" instrumentUnit="arcsec"/>
          <Field name="Chop Qualifier" type="String"
               default="none"
               required="false">
               <ListConstraint name="Parameter">
                    <Choice value="STOP"/>
                    <Choice value="DEBUG"/>
                    <Choice value="RELOAD"/>
                    <Choice value="none"/>
               </ListConstraint>
          </Field>
</Command>
```

Figure 3: IML Description of Telescope Secondary Command

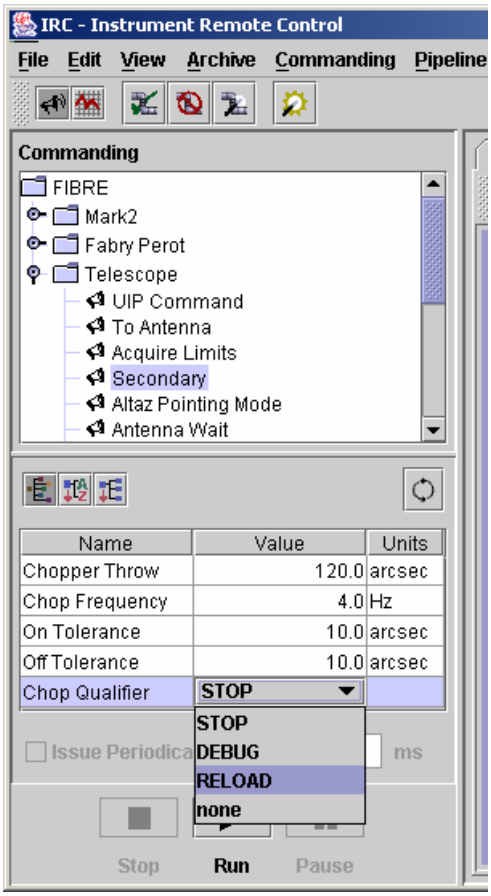

Figure 4: Generated GUI for Secondary Command

Each command has an associated format, specified in the IML, which describes how it should be formatted for transmission to the instrument. Based on our sample Command and Format, see Figure 5, a Secondary Command sent to the Telescope subsystem will look like as follows:

```
SECONDARY 120.0 4.0 10.0 10.0 /RELOAD<cr>
```

```
<Port name="Telescope" portType="TCP" encodingType="RegExp">
     <!—Port parameters go here →
     <Format name="commandFormat" formatType="command">
          <Parameter name="template"
               value="command { white fieldValue } term"/>
          <FormattedInterface name="Command">
               <CommandInterfaceReference reference="CI1"/>
                    <ValueMap name="Secondary" value="SECONDARY" type="String"/>
                    <ValueMap name="none" value="" type="String"/>
                    <ValueMap name="STOP" value="/STOP" type="String"/>
                    <ValueMap name="RELOAD" value="/RELOAD" type="String"/>
                    <ValueMap name="DEBUG" value="/DEBUG" type="String"/>
          </FormattedInterface>
     </Format>
     <TypeFormat name="String" format="%s"/>
     <TypeFormat name="Float" format="%f" />
</Port>
```

Figure 5: IML Description of Telescope Secondary Command Format

While there are many advantages to using IML, one of the most significant is the ability to defer some of the hardware implementation details as long as necessary during the development period. Software often needs to be developed in parallel with the hardware it is to control. Since hardware engineers may need to change various details as their subsystems are integrated, or as new hardware components with different characteristics are manufactured, it is crucial that the software architecture provide a degree of separation between the objects that represent the system and the hardware nuts-and-bolts.

IML enables iterative development because the instrument description is read at runtime. A concrete example of this is the ability to dynamically alter the GUI to reflect a different hardware interface. For example, suppose the hardware engineer decides that he wants to make available a new Telescope Command. He can simply define his new command and its format in the IML file, and the next time the application is started, the new command will appear in the GUI. No new code must be written, nor is recompilation necessary.

4. DISTRIBUTED ARCHITECTURE

Section 3 described the internals of the IRC Framework and how an IML description is used to communicate with an instrument. With a distributed environment we need to take a broader view of an IRC architecture based on multiple instances of the IRC framework. As shown in Figure 6 a single IRC Device can use IML descriptions in two different contexts. The first as outlined in the previous section is a description of the private interface to an instrument that this device will be communicating with. The second context is a description of a device's own public interfaces that other external clients can use to communicate with it.

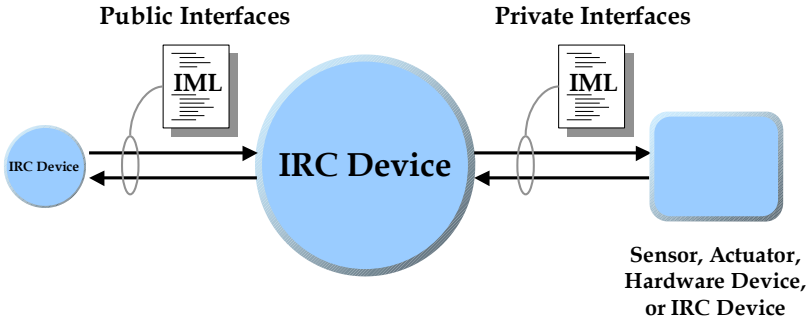

Figure 6: IRC Device External Interfaces

The IRC framework provides the ability to dynamically discover and communicate with other devices anywhere on a network in a peer-to-peer manner[1]. To enable a dynamic discovery and configuration capability for a collection of devices, each IRC instance (referred to as an IRC Device) can advertise and publish information about itself on a virtual network. A virtual network allows devices to communicate and organize independently from the physical network. For scoping and security the virtual network can be divided into virtual peer groups. Devices can join or leave a virtual peer group and thus join or leave the instrument control environment of IRC.

An IRC Device can advertise and publish several public IML interface descriptions. A device may want to split up its public descriptions (commanding vs. data) or publish more than one version (novice vs. expert). This simple capability of dynamically publishing and subscribing to interfaces enables a very flexible self-adapting architecture for monitoring and controlling complex instruments in diverse environments.

4.1. Example Distributed IRC Architecture

The HAWC will be a facility instrument for NASA's SOFIA mission, a Boeing 747SP aircraft modified to accommodate a 2.5m reflecting telescope. The HAWC control and monitor software will be configured in distributed hierarchal peer architecture as illustrated in Figure 7.

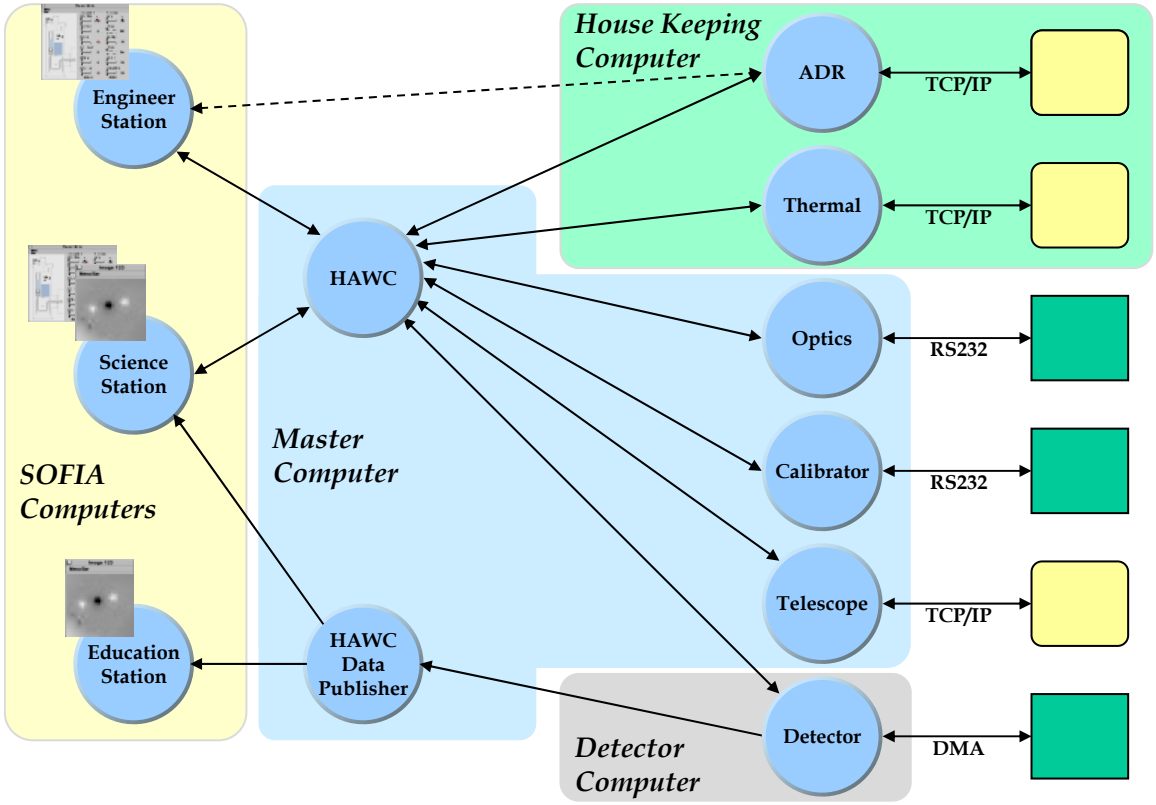

Figure 7: HAWC IRC Device Architecture

Each of the subsystems (ADR, Thermal, Optics, etc.) will have dedicated IRC devices to control them and function as subsystem proxies. The proxies will primarily encapsulate and perform subsystem specific functions and advertise the subsystem on the network to a "HAWC Subsystem" peer group. The type of specific functions that each proxy may perform includes but is not limited to closed loop control, data translation or calibration, and command translation. The

[1] IRC recently transitioned from an in-house developed peer-to-peer framework called WorkPlace to JXTA (see http://www.jxta.org). JXTA defines a set of open protocols for finding and organizing a virtual network of peers.

"HAWC" IRC Device will join the "HAWC Subsystem" peer group as a trusted peer and request the published IML interface for all subsystems. The "HAWC" device will also join a "HAWC Instrument" peer group and publish its IML interface to the group. Astronomers and engineers will be able to start client IRC Devices anywhere on the network, join the "HAWC Instrument" group, and request the public IML description from the "HAWC" device. The "HAWC" device may publish more than one version of the interface depending on the type or authorization of the user.

4.2. Distributed IML Examples

The IML examples in section 3.1, show port definitions with specific IP addresses in the IML for the Telescope IRC Device. Using the distributed architecture of IRC, these descriptions can be much more independent of the physical location of the devices. Figure 8 shows a simplified view of the previous definition. The dynamic discovery mechanisms of IRC are used to find the IML associated with each subsystem. With this approach, the HAWC instrument can be unaware of the physical location of the other peers on the network. It simply knows the names of the peers. Other variants of this allow for IRC to search for all peers in a particular group or for all peers of a particular name, regardless of the group.

```
<Instrument name="HAWC">

     <InstrumentPeer group="HAWC Subsystem" description="Optics"/>
     <InstrumentPeer group="HAWC Subsystem" description="ADR"/>
     <InstrumentPeer group="HAWC Subsystem" description="Thermal"/>
     <InstrumentPeer group="HAWC Subsystem" description="Calibrator"/>
     <InstrumentPeer group="HAWC Subsystem" description="Detector"/>
     <InstrumentPeer group="HAWC Subsystem" description="Telescope"/>
</Instrument>
```

Figure 8: IML description of instrument and subsystems

5. DATA ANALYSIS PIPELINE

After parsing a raw telemetry stream, the Parser publishes the telemetry as Data Objects. Any subscriber in the system can register to receive these Data Objects; however, the IRC framework includes a Data Analysis Pipeline to facilitate the processing of this data. This real-time pipeline is composed of Pipeline Elements that are linked together, i.e. the inputs of one pipeline element are transformed and their output becomes the input to the next element in the pipeline. Examples of pipeline elements are as follows:

- General purpose data processing algorithms, such as parameterized algorithms that apply a scale factor or polynomial function to selected input data
- Instrument specific data processing algorithms
- Data recorders (for archiving data) and players (for reading archives)
- Data visualizations
- Data analysis scripts for autonomous commanding

Pipeline Element types are described using the Pipeline Algorithm Markup Language (PAML), another dialect of XML. With PAML, you must define the following attributes of a Pipeline Element:

- Type name – this is used by IRC to look up a Java class that implements the algorithm.
- Outputs – the characteristics of the output of the Pipeline Element
- Inputs – the characteristics of the input to the Pipeline Element
- Properties – a set of attributes that can be used to configure the Pipeline Element

Based on the PAML description, IRC provides three mechanisms for connecting Pipeline Elements together and setting Pipeline Element properties:

- The default GUI. The GUI presents the algorithm types and allows the user to establish the connection at run-time.

- Pipeline Configuration file. Pipeline configurations can be saved to a file. They can be loaded as part of the startup sequence of IRC or manually through the GUI.
- Scripted control. A script in JPython, Javascript, or other scripting language can be supplied to IRC as part of the startup sequence to establish the pipeline. See section 6 for more information about scripting.

IRC provides many general-purpose algorithms, and aims to make it easy to develop instrument-specific algorithms. Taking advantage of Java's dynamic class loading, the IRC framework does not have to know about the algorithm implementation class until runtime; by referencing the location of the Java byte code, the pipeline manager is able to create instances of pipeline algorithms as needed. Also, by using the Java Native Interface (JNI), algorithms can be implemented in any native language such as C, C++, or FORTRAN, or possibly even IDL.

5.1. Data Visualizations

The IRC software provides several visualizations that can be added to the Pipeline as Pipeline Elements. Visualizations are special purpose Pipeline Elements. This design affords the IRC software some important flexibility. Since a visualization is a Pipeline Element, it may be placed anywhere within the pipeline. This enables a user to view raw data early in the pipeline, or the results of complex calculations performed by a series of Pipeline Elements – or both. The framework does not restrict the user to a single visualization. A user may place several visualizations at various points in the pipeline, providing the ability to monitor data at many stages of analysis simultaneously. The implementation also allows a visualization to publish information, such as statistical or snapshot data, to other parts of the pipeline for archiving or further processing.

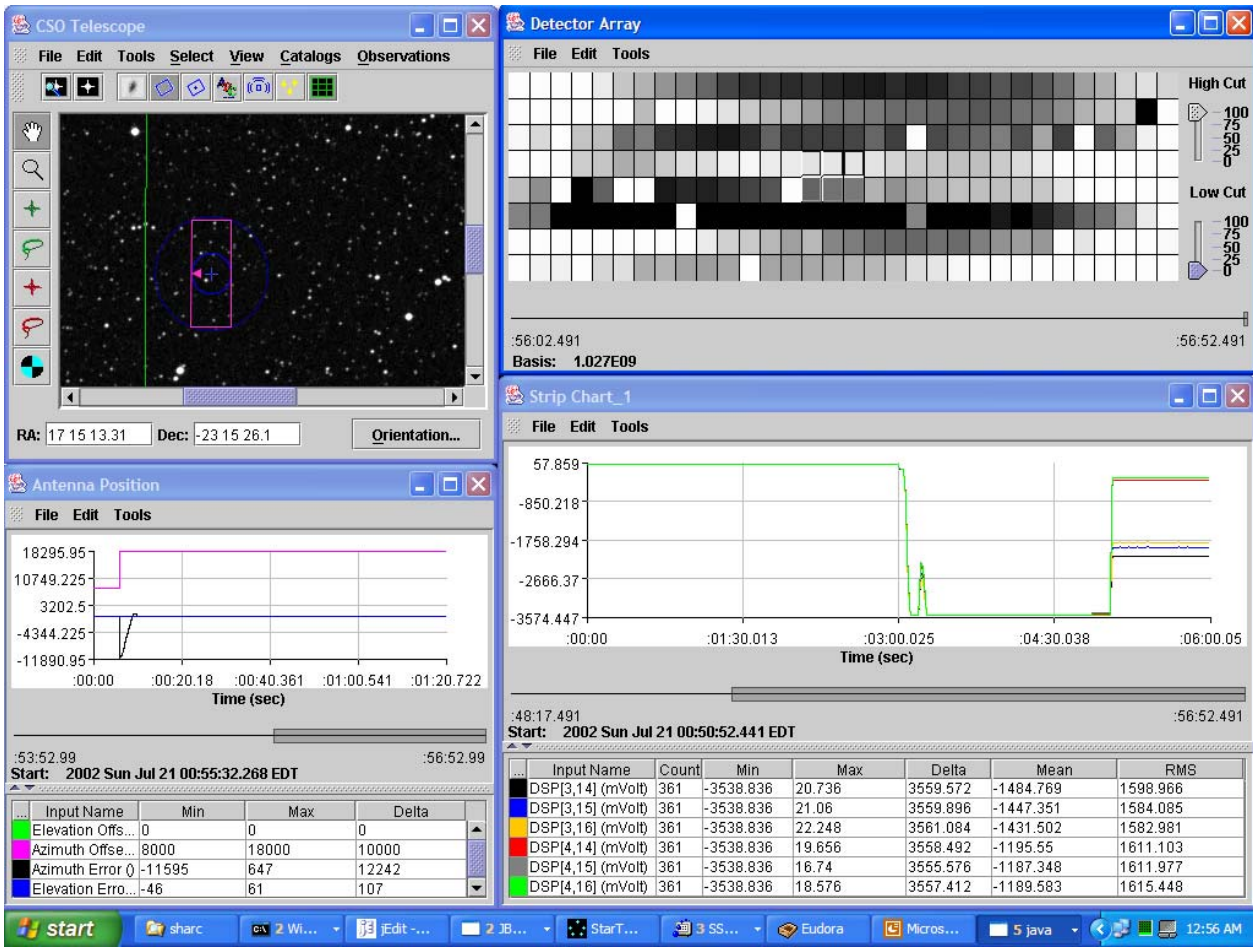

Figure 9: Visualization Examples

6. SCRIPTING

The ability to write scripts to embed in the instrument control software is an important feature of the IRC framework. It provides the user with a way to sequence common tasks. Currently, scripts must be written in Jpython or JavaScript; however, the IRC architecture allows for support of any scripting language that supports the Bean Scripting Framework. JPython is a Java implementation of Python, an interpreted, object-oriented programming language. JPython and Python are free and the source code is available under an Open Source license.

A simple script that strings together a set of instrument commands can be written easily. Such a script is shown in Figure 10. A script can also prompt the user for input, and can add, remove, and configure Pipeline Elements. Support for looping and control flow is included. Using more advanced capabilities of JPython, a script can extend the IRC framework in interesting ways. JPython has full access to all Java packages and Python modules and can extend Java objects. These features have been used to create scripts that implement pipeline algorithms that insert themselves into the pipeline, issue commands based on the analysis of incoming data, and then remove themselves from the pipeline.

```
# File: powerup_detector.py
# Initialize the MkII electronics at detector power-up

from gov.nasa.gsfc.irc.datatypes import BitArray

sendCommand(SPIRE_DETECTOR, "Stop Data")
sendCommand(SPIRE_DETECTOR, "Reset Time Counter")
sendCommand(SPIRE_DETECTOR, "Reset Command Counter")
callProcedure("Initialize Row Address Lookup Table")
callProcedure("Initialize Parameter RAM Lookup Table")
callProcedure("Setup FFCP Frame", BitArray("0001"), 8, 100)
```

Figure 10: Simple Script - Powerup Detector

To make a script available to the system, a fragment of IML must be created that describes the script, its arguments (including data types and valid values), and any documentation for the script. This IML fragment can be added to a library of scripts or to the description of a subsystem to make the script appear as a primitive command to the user. The Command Procedure Manager as shown in Figure 11 can be used to dynamically add and edit a script at runtime and create the IML script fragment.

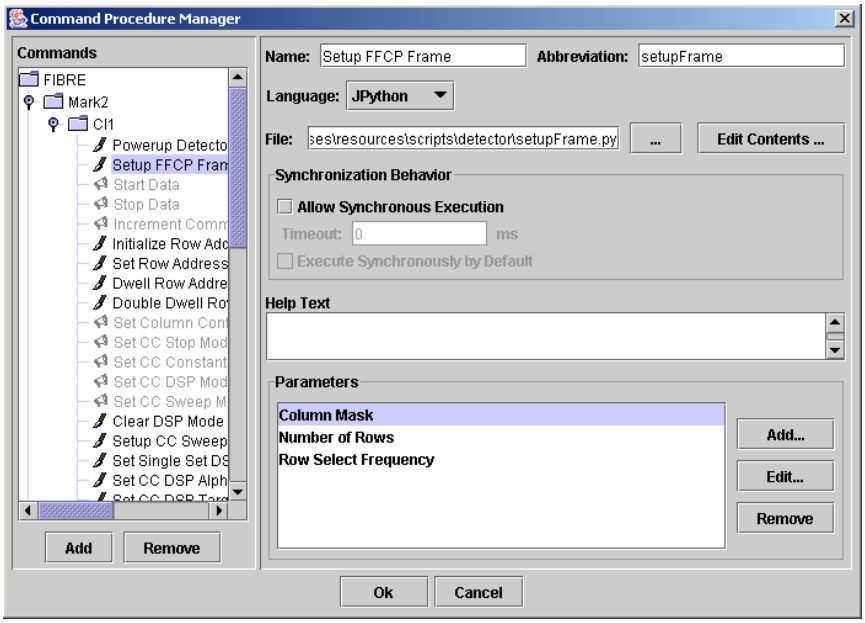

Figure 11: Command Procedure Manager

7. THE VISION: THE FUTURE OF IRC

The IRC framework supports instrument development from early design through operations and maintenance to minimize software development time, minimize development costs, maximize reuse of software components, and maximize flexibility of instrument architectures.

To minimize software development time and to accommodate new instruments, the IRC framework will make it easy to generate large portions of the instrument control application automatically from a formal description of the instrument. This instrument description will be used to create customized GUIs, software component interfaces, specific component configurations, and documentation. Software tools will be created to help instrument designers develop descriptions of instruments. Thus, as new instruments are added to a system, or as specifications for existing instruments are modified, the effort to adapt the software to these changes will be incremental rather than major.

The IRC framework that we envision can be applied to any or all phases of the science life cycle to maximize the science potential of missions. An important enhancement to the IRC framework will be to provide the ability to quickly develop simulations that accurately model instrument operations with whatever degree of fidelity is deemed necessary. Simulations allow many activities to be performed long before instrument development has been completed. Instrument designers can develop, validate, and modify designs quickly and efficiently. Scientists can begin science planning and data analysis algorithm development; data archival, retrieval, and publication scenarios can be worked out; and support staff can begin training for instrument operations very early in the program. The underlying open software infrastructure that we envision will enable single and multiple discipline behavior models to be assembled and synthesized into instrument simulations to facilitate the rapid design and development of next generation instrument designs and operations.

The IRC framework of the future will maximize the ability to incorporate emerging technologies. The design supports a high degree of configurability, allowing it to be tuned for specific observatories or other factors. Processes can be run on a single computer or on multiple heterogeneous computers, ranging from small, low cost hardware components to high-end workstations. Processes can be run either at the observatory or remotely over the Internet (or both). This provides an instrument development team the flexibility to use the hardware components that best fit the operating environment and instrument requirements. The framework will support the cross-platform migration of functions and necessary reconfiguration if these requirements change. This flexibility enables a new design in which small, embedded software components are placed at the point of origin of the generated data (smart sensors) and at the point of device control (smart actuators). The envisioned configurable software framework will enable these software solutions to be easily developed, enhanced, maintained, and reused for different devices, different instruments, and different domains.

7.1. Next Steps

The near-term efforts of the IRC team fall into two main categories: (1) Enhancing and improving the general IRC framework, including incorporating lessons learned from applying it to existing NASA instruments; and (2) using the IRC framework to develop the instrument control and monitoring software for HAWC, SAFIRE, and SHARCII.

Currently, the IRC framework provides a default GUI that enables a user to issue every command and script defined in the IML. This type of GUI is appropriate for an instrument's engineering test phase, but is not as useful for general instrument operations. We have always intended to have a way to customize the GUI. We have prototyped using Java's 1.4 long-term persistence API in order to allow instrument teams to specify customized GUIs in XML. The next step is to incorporate this prototyping effort into the IRC framework.

In applying the IRC software to one or more instruments, the activities involved include the following:

1. Develop the instrument description. This involves mapping the ICDs from each of the instrument subsystem teams, including the ICD for the telescope, into an IML description for the subsystem.
2. Develop the instrument-specific real-time pipeline algorithms.
3. Develop custom GUIs for the various instrument users.
4. Develop new visualizations.

5. Develop instrument-specific scripts.
6. Develop any instrument-specific delegates (for special purpose parsing or response handling).

ACKNOWLEDGEMENTS

Support from the following people has proven to be invaluable: Dr. Rick Shafer, Dr. Dominic Benford – NASA GSFC FIBRE / SAFIRE team; Dr. Robert Loewenstein – Yerkes Observatory HAWC team; Dr. Darren Dowell – Caltech SHARCII team.

REFERENCES

1. T. Ames, L. Koons, K. Sall, (1999). *Instrument Remote Control* [Online]. NASA Goddard Space Flight Center. Available: http://pioneer.gsfc.nasa.gov/public/irc/ [2000, February 20]
2. T. Ames, L. Koons, (1999). *IRC Presentations, Publications and Milestones* [Online]. NASA Goddard Space Flight Center. Available: http://pioneer.gsfc.nasa.gov/public/irc/IRC-presentations.html [2000, February 20]
3. T. Ames, L. Koons, K. Sall, (1999). *Instrument Markup Language* [Online]. NASA Goddard Space Flight Center. Available: http://pioneer.gsfc.nasa.gov/public/iml/ [2000, February 20]
4. J. Breed, (1999). *Code 588 Website – Advanced Architectures Automation Branch* [Online]. NASA Goddard Space Flight Center. Available: http://aaaprod.gsfc.nasa.gov/website/WebSite.cfm [2000, February 20]
5. Sterling Software, (1999). *SOFIA Homepage - The latest in Airborne Astronomy* [Online]. NASA Goddard Space Flight Center. Available: http://sofia.arc.nasa.gov/ [2000, February 20]
6. *HAWC* [Online]. Yerkes Observatory. Available: http://astro.uchicago.edu/hawc/hawc.htm [2000, February 20]
7. T. Ames, L. Koons, K. Sall (1999). *Submillimeter And Far Infrared Experiment* [Online]. NASA Goddard Space Flight Center. Available: http://pioneer.gsfc.nasa.gov/public/safire/ [2000, February 20]
8. T. Ames, L. Koons, K. Sall (1999). *Spectral and Photometric Imaging REceiver* [Online]. NASA Goddard Space Flight Center. Available: http://pioneer.gsfc.nasa.gov/public/spire/ [2000, February 20]
9. *The Source for Java™ Technology* [Online]. Sun Microsystems. Available: http://java.sun.com/ [2000 February 20]
10. D. Connolly (2000). *Extensible Markup Language (XML)* [Online]. World Wide Web Consortium. Available: http://www.w3.org/XML/ [2000, February 22]
11. J. Hosler, M. Brandt, K. Hughes (2000). *Advanced Visual Tools and Architectures* [Online]. NASA Goddard Space Flight Center. Available: http://wave.gsfc.nasa.gov/avatar/ [2000, February 20]
12. *JPython Home* [Online]. The Python Consortium. Available: http://www.jpython.org/ [2000, February 20]